EnTagRec: An Enhanced Tag Recommendation System for Software Information Sites

Shaowei Wang*, David Lo*, Bogdan Vasilescu†, and Alexander Serebrenik†

*School of Information Systems, Singapore Management University, Singapore

†Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands

Email: {shaowei.wang.2010, davidlo}@smu.edu.sg, {b.n.vasilescu, a.serebrenik}@tue.nl

Abstract—Software engineers share experiences with modern technologies by means of software information sites, such as STACK OVERFLOW. These sites allow developers to label posted content, referred to as *software objects*, with short descriptions, known as *tags*. However, tags assigned to objects tend to be noisy and some objects are not well tagged.

To improve the quality of tags in software information sites, we propose ENTAGREC, an automatic tag recommender based on historical tag assignments to software objects and we evaluate its performance on four software information sites, STACK OVERFLOW, ASK UBUNTU, ASK DIFFERENT, and FREECODE. We observe that that ENTAGREC achieves *Recall@5* scores of 0.805, 0.815, 0.88 and 0.64, and *Recall@10* scores of 0.868, 0.876, 0.944 and 0.753, on STACK OVERFLOW, ASK UBUNTU, ASK DIFFERENT, and FREECODE, respectively. In terms of *Recall@5* and *Recall@10*, averaging across the 4 datasets, ENTAGREC improves TAGCOMBINE, which is the state of the art approach, by 27.3% and 12.9% respectively.

I. INTRODUCTION

The growing online media has significantly changed the way people communicate, collaborate and share information with one another [35]. This is also true for software developers, who create and maintain software by standing on the shoulders of others [31], reuse components and libraries originating from Open Source repositories (e.g., FREECODE), and forage online for information that will help them in their tasks [8]. When foraging for information, developers often turn to programming question and answer (Q&A) communities such as STACK OVERFLOW, ASK UBUNTU and ASK DIFFERENT. Such sites supporting communication, collaboration and information sharing among developers are known as *software information sites*, while their contents (e.g., questions and answers, project descriptions)—as *software objects* [40].

Typically, tags are short labels not more than a few words long, provided as metadata to software objects in software information sites. Users can attach tags to various software objects, effectively linking them and creating topic-related structure. Tags are therefore useful for providing a soft categorization of the software objects and facilitating search for relevant information. To accommodate new content, most software information sites allow users to create tags freely. However, this freedom comes at a cost, as tags can be idiosyncratic due to users' personal terminology [16]. As tagging is inherently a distributed and uncoordinated process, often similar objects are tagged differently [40]. For example, in STACK OVERFLOW the tags `xmlparser`, `xml-parser` and `xmlparsing` are all used to describe a parser of an XML file. Idiosyncrasy reduces the usefulness of tags, since related objects are not linked together by a common tag and relevant information becomes more difficult to retrieve. Furthermore,

some software information sites (e.g., STACK OVERFLOW) require users to add tags at the time of posting a question, even if they are unfamiliar with the tags in circulation at that time. Due to differences in personal terminology and tagging purpose, it is often difficult for users to select appropriate tags for their content. Having a tag recommendation system that can suggest tags to a new object (e.g., based on how other similar objects have been tagged in the past) could (i) help users select appropriate tags easily and quickly, and (ii) in time help homogenize the entire collection of tags such that similar objects are linked together by common tags more frequently. To illustrate the importance of tags for the well functioning of a software information site, note the more than 4000 questions related to tags on the META STACK OVERFLOW[1], as opposed say to only the about 1000 related to user interface.

For this purpose, we propose an automatic tag recommendation system called ENTAGREC. ENTAGREC learns from historical software objects and their tags, and recommends appropriate tags for new objects. ENTAGREC borrows from two opposite yet complementary lines of thought in the statistics community, Bayesian and frequentist [28], each with its own advantages [5], [13]. In trying to combine the advantages of both, ENTAGREC consists of two inference components. Our Bayesian inference component (BIC) models a software object as *a probability distribution* of tags, and a tag as *a probability distribution* of words that appear in the software objects that are tagged by it. Our BIC is built on top of a state-of-the-art Bayesian inference technique (Labeled LDA [27]), creating a unified mixture model over the distribution of all tags to measure the likelihood of a tag to be assigned to a software object. Our frequentist inference component (FIC) considers a software objects as *a set* of tags attached to it, and a tag as *a set* of words appearing in objects that are tagged by it. We propose an extended FIC which removes unrelated words in software objects and leverages the parts-of-speech (POS) tagger [33] to reduce noise, and the spreading activation algorithm [11] to increase the number of correct tags successfully inferred.

Given a software object to be tagged, each of these two components outputs a probability score for every tag, which indicates the likelihood of the tag to be assigned to the software object. We combine these two components by taking a weighted sum of the probability scores as the final scores of the tags. Tags are then ranked based on these final scores and are recommended to users. Our approach addresses the limitation of TAGCOMBINE [40], the state-of-the-art tag recommender system for software information sites, which creates multiple models by performing one-versus-rest analysis instead of using

---

[1]http://meta.stackexchange.com/questions/tagged/tags

a unified mixture model and neither removes unrelated words nor leverage POS tagger to reduce noise.

We evaluate our approach on datasets from four popular software information sites, STACK OVERFLOW, ASK UBUNTU, ASK DIFFERENT, and FREECODE. STACK OVERFLOW and FREECODE have been used previously to evaluate TAGCOMBINE. Our experimental results show that ENTAGREC achieves *Recall@5* scores of 0.805, 0.815, 0.88 and 0.64, and *Recall@10* scores of 0.868, 0.876, 0.944, 0.753 on STACK OVERFLOW, ASK UBUNTU, ASK DIFFERENT, and FREECODE, respectively. Compared with TAGCOMBINE, averaging across the four software information sites, ENTAGREC improves TAGCOMBINE by 27.3% in terms of *Recall@5* and 12.9% in terms of *Recall@10*. Our main contributions are:

- We propose ENTAGREC, a novel automatic tag recommendation system for software information sites. ENTAGREC composes a state-of-the-art Bayesian inference technique (Labeled LDA), and an enhanced frequentist inference technique that leverages a POS tagger and the spreading activation algorithm.
- We extend the empirical study by Xia et al. [40] to evaluate TAGCOMBINE, by investigating datasets from four popular software information sites. Our study shows that ENTAGREC can achieve high recall, especially for STACK OVERFLOW, ASK UBUNTU, and ASK DIFFERENT. ENTAGREC also improves upon TAGCOMBINE by a decent margin of 27.3% in terms of *Recall@5* and 12.9% in terms of *Recall@10*.

The rest of this paper is organized as follows. We provide more background on tags in several software information sites in Section II. We present the high-level architecture of ENTAGREC in Section III, followed by detailed descriptions of the Bayesian and frequentist inference components in Sections IV and V, and the specifics of how to integrate the two components in Section VI. We present our evaluation results in Section VII. Finally, we highlight related work in Section VIII and conclude in Section IX.

II. TAGS IN SOFTWARE INFORMATION SITES

To facilitate navigation, search and filtering, content are marked with descriptive terms [16], known as tags: libraries associate books with authors' names and keywords, while scientific publishers require the authors to choose keywords themselves. In the digital world, tags can be used, e.g., to annotate weblog posts or shared links. Numerous software information sites employ tags, e.g., SourceForge[2] for code projects, Eclipse MarketPlace[3] for plugins, Snipplr[4] for code fragments, or STACK OVERFLOW for questions.

An example STACK OVERFLOW question is presented in Figure 1. The question pertains to the creation of an Eclipse plugin and it has two tags, representing the technical context of the question (`eclipse`) and a specific subject area (`eclipse-plugin`). Figure 2 shows the FREECODE description of Apache Ant: in addition to the textual description, two general tags are present, `Software Development` (describing the general domain of Apache Ant) and `Build Tools` (indicating a more specific functionality of Apache Ant, namely building Java programs).

---

[2]http://sourceforge.net/
[3]http://marketplace.eclipse.org/
[4]http://snipplr.com/

Fig. 1. A Question in STACK OVERFLOW

Fig. 2. A Project in FREECODE

Comparing Figures 1 and 2 we observe that while the basic purpose of tagging—to facilitate navigation, search and content filtering through the association of related contents via linked descriptive terms—is common to both, specific policies how the tags should be used differ from site to site. For instance, Eclipse MarketPlace requires a user to choose five out of 48 predefined categories and one out of three markets, as well as allows her to provide additional tags not subject to content restrictions. In contrast, Stack Exchange websites do not distinguish between different kinds of tags but restrict the total number of tags given to a question. Moreover, while tags on Eclipse MarketPlace are subject to moderation, i.e., they can be modified only by the author or by a moderator, modifications of tags on Stack Exchange sites can be proposed by any user and are carried out by more experienced ones.

Most software information sites allow users to provide "free text tags". Not being subject to the formal requirements of the sites, such tags can be expected to represent user intent in a more flexible way. However, tagging becomes a distributed and uncoordinated process, introducing different tags for similar objects, which might persist despite moderation or the ongoing correction efforts.For example, questions on STACK OVERFLOW entitled "SIFT and SURF feature extraction implementation using MATLAB"[5] and "Matlab implementation of Haar feature extraction"[6] are both related to image feature extraction but only the second one is labeled with the corresponding tag, i.e., `feature-extraction`.

III. OVERALL ARCHITECTURE

In this section we describe the overall architecture of our ENTAGREC approach. ENTAGREC contains four processing components: Preprocessing Component (PC), Bayesian Inference Component (BIC), Frequentist Inference Component (FIC) and Composer Component (CC). Input software objects are processed by PC to generate a common representation. These textual documents are then input to the two main processing engines, namely BIC and FIC. BIC models a software objects as a *probability distribution of tags*, and a tag as a probability distribution of words that appear in the software objects that are tagged by it. FIC considers a software objects as *a set of tags* that are attached to it, and a tag as *a*

---

[5]http://stackoverflow.com/q/5550896
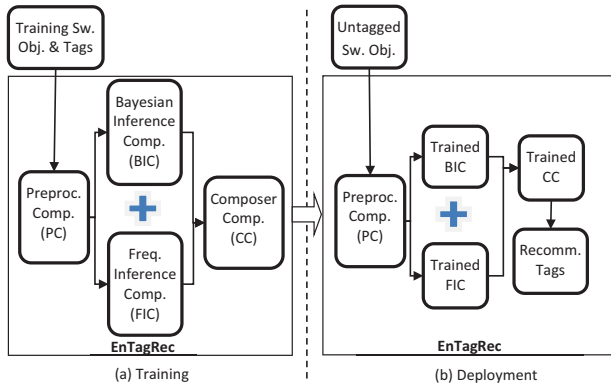[6]http://stackoverflow.com/q/2058138

Fig. 3. ENTAGREC Architecture

set of words that appear in objects that are tagged by it. CC combines the complementary BIC and FIC components.

ENTAGREC works in two phases, a training phase and a deployment phase, as shown in Figure 3(a) and (b), respectively. In the training phase, ENTAGREC trains several of its components using training software objects and corresponding tags. In the deployment phase, the trained ENTAGREC is used to recommend tags for untagged software objects.

The common component in the training and deployment phase is PC that converts each software object into a bag (or multiset) of words. The PC starts from the textual description of a software object and performs tokenization, identifier splitting, number removal, stop word removal, and stemming. Tokenization breaks a document into word tokens. Identifier splitting breaks a source code identifier into multiple words. We use a simple Camel Casing splitter [2], e.g., the identifier "getMethodName" will be split into "get", "method", and "name". Number removal deletes numbers. Stop word removal[7] deletes words that are used in almost every document and therefore, carry little document-specific meaning, e.g., "the", "is", etc. Finally, stemming reduces words to their root form. We use the Porter stemming algorithm [25].

In the training phase, BIC, FIC, and CC are trained based on the training data. BIC takes the bag of words representation of the software objects and their corresponding tags to train itself. The result is a statistical model which takes in a bag of words representing a software object and produces a ranked list of tags along with their probabilities of being related to the input software object. FIC also processes the bag of words representations and the corresponding tags to train itself and it produces a statistical model (albeit in a different way than BIC) which also takes in a bag of words and outputs a ranked list of tags with their probabilities. CC learns two weights for BIC and FIC to generate a near-optimal combination of these two components from the training data.

After ENTAGREC is trained, it is used in the deployment phase to recommend tags for untagged objects. For each such object, we first use PC to convert it to bags of words. Next, we feed this bag of words to the trained BIC and FIC. Each of them will produce a list of tags with their likelihood scores. CC will compute the final likelihood score for the tags based

---

on the weights that it has learned in the training phase. The top few tags with the highest likelihood scores will be output as the predicted tags of the input untagged software object.

The following sections detail each of the three major components of ENTAGREC, BIC, FIC, and CC.

IV. BAYESIAN INFERENCE

The goal of BIC is to compute the probabilities of various tags given a bag of words representing a software object using Bayesian inference. Given a tag t and a software object o, BIC computes the probability of t being assigned to o given the words $\{w_1, \ldots, w_n\}$ that appear in o. Mathematically, this is denoted as $P(t|w_1 \ldots w_n)$. Using the Bayes theorem [14], this probability can be computed as:

$$P(t|w_1 \ldots w_n) = \frac{P(w_1 \ldots w_n|t) \times p(t)}{p(w_1 \ldots w_n)}$$

The probabilities on the right hand side of the above equation can be estimated based on a training data.

A state-of-the-art Bayesian inference algorithm is Latent Dirichlet Allocation (LDA) [7]. LDA has been shown effective to process various software engineering data for various tasks, e.g., [3], [4], [23], [24]. LDA takes in a set of documents and a number of topics K and outputs the probability distribution of topics per document. Our problem can be easily mapped to LDA where a document corresponds to a software object and a topic corresponds to a tag. Using this setting, LDA outputs the probability distribution of tags for a software object.

However, LDA is an unsupervised learning algorithm. It does not take as input any training data and it is not possible to pre-define a set of tags as the target topics to be assigned to documents. Fortunately, recent advances in the natural language processing community introduced extensions to LDA, such as Labeled LDA (L-LDA) [27]. For L-LDA, the labels can be predefined and a training set of documents can be used to train the LDA such that it will compute the probability distribution of topics, coming from a predefined label set (tags, in our case), for a document (a software object, in our case), based on a set of labeled training data. In this work, we use L-LDA as the basis for the Bayesian inference component.

BIC works on two phases: training and deployment. In the training phase, BIC takes as input a set of bags of words representing software objects and their associated tag. These are used to train an L-LDA model. In the deployment phase, given a bag of words corresponding to a software object, the trained L-LDA model is used to infer the set of tags for the input software object along with their probabilities. In the end, the top $K_{Bayesian}$ inferred tags for the object will be output and fed to the Composer Component (CC).

V. FREQUENTIST INFERENCE

FIC computes the probability that a software object is assigned a particular tag based on the words that appear in the software object, taking into account the *number* of words that appear along with the tag in software objects in a training set. Section V-A describes our basic approach and several extensions are presented in Section V-B. Unless otherwise stated hereafter, FIC refers to the extended approach.

---

[7]We use the list of stopwords from http://www.textfixer.com/resources/common-english-words.txt

A. Basic Approach

Consider software object o with n words: $\{w_1, w_2, \ldots, w_n\}$. Assuming the words in o to be independent, the probability of o to be assigned tag t (i.e., $P(o,t)$) is:

$$P(o,t) = \prod_{i=1}^{n} P(t|w_i)$$

The above probabilities ($P(t|w_i)$ for $1 \leq i \leq n$) can be estimated from the training data. However, the probability will be zero if any one of the words used in o does not appear in any software objects in the training data. To address this issue, we compute the following weight instead of the actual probability:

$$W(o,t) = \sum_{w_i \in o} P(t|w_i)$$

If the probabilities are non-zero, as probability $P(o,t)$ increases, weight $W(o,t)$ will increase as well.

Furthermore, to reduce the computational cost, given a training set of tagged software objects *TRAIN*, each of the probabilities, i.e., $P(t|w_i)$ for $1 \leq i \leq n$, is estimated as:

$$P'(t|w_i) = \begin{cases} 1, & \exists\, o \in \textit{TRAIN},\ o \text{ contains } w_i \text{ \& tagged with } t \\ 0, & \text{Otherwise} \end{cases}$$

In effect, we compute the following weight:

$$W'(o,t) = \sum_{w_i \in o} P'(t|w_i)$$

The higher the weight $W'(o,t)$ is, the more representative FIC deems tag t is to software object o. We further normalize weight $W'(o,t)$ to unit range, and treat it as a proxy to the original probability of an object o to be assigned tag t.

B. Extended Approach

There are several problems with the basic approach. First, often not all words in a software object are related to the tags that are assigned to the software object. Although the pre-processing component (PC) has removed stop words, still many non stop words are unrelated to software object tags, thus need to be removed. Second, we have a data sparsity problem, since many tags are not used in many software objects in the training set. Thus, often a tag is not characterized by sufficiently many words. To address this problem, we leverage the relationships among tags to recommend additional associated tags to an input untagged software object.

1) Removing Unrelated Words with POS Tagger: One problem in estimating the probabilities $P(t|w_i)$ is that not all words that appear in a software object are related to the tags. We use the example in Figure 1 to illustrate this. Words "need" and "work" are not stop words but they are unrelated to the tags `eclipse` and `eclipse-plugin`. Thus, there is a need to filter out these unrelated words before we estimate the probabilities.

We observe that nouns and noun phrases are often more related to the tags than other kinds of words. Past studies have also found that nouns are often the most important words [9], [29]. Thus, in this extension, we remove all words except nouns and noun phrases. To identify these nouns and noun phrases, we use the Part-Of-Speech (POS) Tagger [33] to infer the POS of each word in the representative bag of words of a software object. In this paper, we user the Stanford Log-linear Part-Of-Speech Tagger.[8] To illustrate this extension, consider the words that appear in the software object shown in Figure 1. After this step, only the words "tutorial", "eclipse", "plugin", "interface", "function", "java", "app" remain.

Note that we only did this for FIC and not BIC as L-LDA assigns different probabilities to words that are associated to a topic (i.e., a tag). Unrelated words will receive low probabilities. In FIC, the words that appear in objects tagged with tag t are treated as equally important. Thus, we only perform this extended processing step for FIC.

We refer the basic approach extended by this processing step as *FrePOS*. Given an untagged software object, *FrePOS* outputs the top $K_{Frequentist}$ tags.

2) Finding Associated Tag with Spreading Activation: Due to the data sparseness problem, the tags inferred by *FrePOS* might miss some important tags that are not adequately represented in the training data. To alleviate the data sparseness problem, we leverage the relationships among tags to find additional associated tags to those inferred by *FrePOS*.

To infer these associated tags, we use a technique named spreading activation [11]. Spreading activation takes as input a network containing weighted nodes that are connected with one another with weighted edges, and a set of starting nodes. Initially, all nodes except the starting nodes are assigned weight 0. Spreading activation then processes the starting nodes, one at a time. For each starting node, it spreads (or propagates) the node's weight to its neighboring nodes which are at most *MH* hops away from it (where *MH* is a user-defined threshold). At the end of the process, we output all nodes with non zero weights and their associated weights. In our context, the network is a tag network, the starting nodes are the nodes corresponding to tags returned by *FrePOS*, and the weights of these starting nodes are the probabilities assigned to the corresponding tags by *FrePOS*.

To perform spreading activation, we first need to construct a network of tags. Each node in the network corresponds to a tag, and each edge connecting two nodes in the network corresponds to the relationship between the corresponding tags. The weight of each edge measures how similar two tags are. We measure this based on the co-occurrence of tags in software objects in the training set. Consider a set of tags where each of them is used to label at least one software object in the training set. Mathematically, we denote this set as: $Tags = \{t_1, t_2, t_3, ..., t_k\}$, where k is the total number of unique tags. We denote an edge between two tags t_i and t_j as e_{t_i, t_j}. The weight of e_{t_i, t_j} depends on the number of software objects that are tagged by t_i and t_j in the training set. It can be calculated as follows:

$$weight(e_{t_i, t_j}) = \frac{|Doc(t_j) \bigcap Doc(t_i)|}{|Doc(t_i) \bigcup Doc(t_j)|}$$

where $Doc(t_i)$ and $Doc(t_j)$ are the objects tagged with t_i and t_j, respectively, and $|S|$ denotes cardinality of the set S.

The edge connecting two tags is assigned a higher weight if they appear together more frequently, which means they are more associated with each other. We denote a set of edges connecting pairs of nodes as *Links*. The tag network is then a graph *TN* defined as (*Tags*, *Links*). Given a tag t, we denote

---

[8]http://nlp.stanford.edu/software/tagger.shtml

Algorithm 1 Find Associated Tags Algorithm

1: **FindAssociatedTags**
2: **Input:**
3: *TN*: Tag network
4: *SST*: Set of starting tags
5: *MH*: Maximum hop
6: **Output:** Set of candidate tags
7: **Method:**
8: Initialize the weight of each tag in *TN* with 0
9: **for** each tag t in *SST* **do**
10:     Set *weight* (*TN*[*t*]) = Probability of tag t inferred by *FrePOS*
11: **end for**
12: **for** each tag t in *SST* **do**
13:     Call SpreadingActivation(*TN*, *TN*[*t*], 0, *MH*)
14: **end for**
15: **return** $SST \cup \{t \,|\, weight(TN[t]) > 0\}$

Algorithm 2 Spreading Activation For A Node

1: **SpreadingActivation**
2: **Input:**
3: *TN*: Tag network
4: *N*: Current node
5: *CH*: Current hop
6: *MH* Maximum hop
7: **Method:**
8: **if** $CH > MH$ **or** $weight(TN[T]) = 0$ **then**
9:     **return**
10: **end if**
11: **for** each node N' that is directly connected to N **do**
12:     Set $w = weight(N) \times weight(E(N', N))$
13:     **if** $weight(N') < w$ **then**
14:         $weight(N') = w$
15:         SpreadingActivation(*TN*, N',*CH*+1,*MH*)
16:     **end if**
17: **end for**

---

the node in TN corresponding to t as $TN[t]$. Given a node n and an edge $E(n_1, n_2)$, we denote their weights as *weight*(n) and *weight*$(E(n_1, n_2))$, respectively.

The pseudocode of our approach to infer associated tags from the initial set of tags returned by *FrePOS* is shown in Algorithm 1. It takes as input a tag network *TN* constructed from all tags in the training data, a set of starting tags *SST* returned by *FrePOS*, and a threshold *MH* that restricts the weight propagation to a maximum number of hops. Our algorithm first initializes the weights of nodes corresponding to tags in the set of starting tags with the probabilities returned by *FrePOS*, and it sets the weights of other nodes to 0 (Lines 8–11). For each starting tag, our algorithm then performs spreading activation starting from the corresponding node in the tag network by calling the procedure `SpreadingActivation` (Lines 12–14). Finally, the algorithm outputs all nodes in the set of starting tags, along with the associated tags, which correspond to nodes in *TN* whose weights are larger than zero (Line 15).

The procedure `SpreadingActivation` spreads the weight of a node to its neighbors. It takes as input a tag network *TN*, a starting node N, the current hop *CH*, and the maximum hop *MH*. The procedure first checks if it needs to propagate the weight of node N—it only propagates if the current hop *CH* does not exceed the threshold *MH*, and the weight of the current node is larger than zero (Lines 8–10). It then iterates through nodes N' that are directly connected to N (Lines 11–17). For each of such nodes, we compute a weight w which is a product of the weight of node N and the weight of the edge linking N to N' (Line 12). If the weight of node N' is less than w, we assign w as the weight of node N' (Lines 13–14). The procedure then tries to propagate the weight of N' to its neighbors by a recursive call to itself (Line 15).
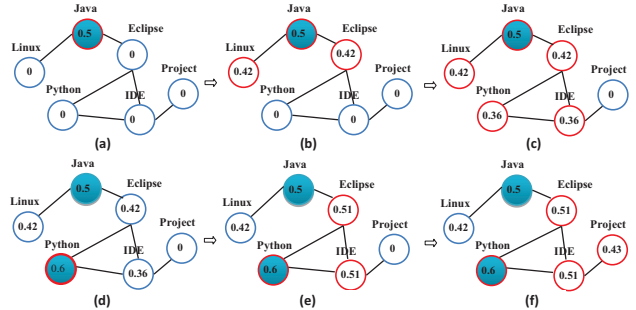

Fig. 4.   Finding Associated Tags Using Spreading Activation: An Example

Example Consider a set of starting tags $SST = \{$JAVA $= 0.5$, PYTHON $= 0.6\}$ output by *FrePOS*, a tag network *TN* shown in Figure 4 and a threshold $MH = 2$. Let us assume the weights of all edges in the tag network are 0.85. At the beginning, our approach initializes the weight of the node corresponding to tag JAVA in *TN* with 0.5 (Figure 4(a)). Then, the weight of node JAVA is propagated to its neighbors LINUX and ECLIPSE and their weights are both updated to 0.42 (Figure 4(b)). The weight is recursively propagated to all neighbors of node JAVA of distance *MH* hops or less (Figure 4(c)). Then, our approach processes tag PYTHON, node PYTHON's weight is updated to 0.6, which is the weight of tag PYTHON output by *FrePOS*. (Figure 4(d)). Our approach then propagates the weight of node PYTHON to its neighbors. If a neighbor's weight is lower than that which is propagated from PYTHON, the original weight is replaced with the new weight. Otherwise, the original weight remains unchanged. Thus, the weights of ECLIPSE and IDE are updated to 0.51 (0.51 exceeds 0.425, the current weights of these tags, Figure 4(e)). The weight of node PROJECT is updated to 0.43 (Figure 4(f)). Finally, the tags JAVA $= 0.5$, PYTHON $= 0.6$, ECLIPSE $= 0.51$, IDE $= 0.51$, LINUX $= 0.42$, PROJECT $= 0.3$ will be output.

The spreading activation process requires a parameter *MH* (maximum hop); by default, we set the parameter *MH* to 1, as the complexity of spreading activation is exponential to the value of *MH*. At the end, our FIC component outputs candidate tags that are output by *FrePOS* and the associated tags that are output by the spreading activation procedure described above. These tags are input to the composer component (CC).

Note that we only apply this spreading activation step to FIC and not BIC. L-LDA used in BIC is more robust than *FrePOS* to the data sparsity problem. We find that the application of this step to BIC does not improve its effectiveness.

VI.   COMPOSER COMPONENT

Given a target software object o, both BIC and FIC produce a list of tags along with their probabilities. The composer component combines the two lists of tags into a unified list of tags along with a set of updated probabilities.

Given a software object o and a tag t, we define the EN-TAGREC ranking score as ENTAGREC$_o(t)$ as follows:

$$\text{ENTAGREC}_o(t) = \alpha \times B_o(t) + \beta \times F_o(t) \qquad (1)$$

where $B_o(t)$ and $F_o(t)$ are the probabilities of tag t computed by BIC and FIC respectively, and $\alpha, \beta \in [0, 1]$ are the weights the composer component assigns to BIC and FIC, respectively.

Algorithm 3 Weight Tuning Algorithm

```
1: TuneWeights
2: Input:
3:   TO: Training Tagged Software Objects
4:   EC: Evaluation Criterion
5:   Tags^B: Set of tags inferred by BIC
6:   Tags^F: Set of tags inferred by FIC
7: Output:
8:   α and β
9: Method:
10: Set α = 0, β = 0
11: for each α from 0 to 1, each step increases α by 0.1 do
12:    for each β from 0 to 1, each step increases β by 0.1 do
13:       for each object o in TO do
14:          for each tag t in Tags^B ⋃ Tags^F do
15:             Compute ENTAGREC_o(t) according to Eq. 1
16:          end for
17:          Sort tags based on their ENTAGREC scores (desc. order)
18:          Evaluate the effectiveness of α and β on o based on EC
19:       end for
20:       Evaluate the effectiveness of α and β on TO based on EC
21:    end for
22: end for
23: return  the best α and β based on EC
```

To automatically tune α and β, we use a set of training software objects and employ grid search [6]. The pseudocode of our weight tuning procedure is shown in Algorithm 3. The weight tuning procedure takes as input the set of training software objects *TO*, an evaluation criteria *EC*, and the two sets of tags returned by BIC and FIC (along with their probabilities). Our tuning procedure initializes α and β to 0 (Line 10). Then, it incrementally increases the value of α and β by 0.1 until they reach 1.0 (Lines 11–12). For each combination of α and β and each software object o in *TO*, our tuning procedure computes the ENTAGREC scores for each tag returned by BIC and FIC (Lines 13–16). Then tags are ordered based on their ENTAGREC scores (Line 17). This is the ranked list of tags that are recommended for o. Next, our tuning procedure evaluates the quality of the resulting ranking based on particular α and β values using *EC* (Line 18). The process is repeated for all objects in *TO* and again the quality of the resulting ranking is evaluated using *EC* (Line 20). The process continues until all combinations of α and β have been exhausted and our tuning procedure finally outputs the best pair of α and β based on *EC* (Line 23).

Various evaluation criteria can be used in our weight tuning procedure. In this paper, we make use of *Recall@k* which have been used as the evaluation criteria in many past tag recommendation studies, e.g., [1], [41]. *Recall@k* was also used in the previous state-of-the-art study on tag inference for software information sites [40]. Definition 1 defines *Recall@k*.

Definition 1: Consider a set of n software objects. For each object o_i, let the set of its correct (i.e., ground truth) tags be $Tags_i^{correct}$. Also, let $Tags_i^{topK}$ be the top-k ranked tags that are recommended by a tag recommendation approach for o_i. *Recall@k* for n is given by:

$$Recall@k = \frac{1}{n} \sum_{i=1}^{n} \frac{|Tags_i^{topK} \bigcap Tags_i^{correct}|}{|Tags_i^{correct}|}$$

In the training phase, the composer component learns the two weights α and β following the procedure given above. In the deployment phase, the composer component combines the recommendations made by BIC and FIC by computing the ENTAGREC scores using Equation 1 for each recommended tag. It then sorts the tags based on their ENTAGREC scores (in descending order) and outputs the top-k ranked tags.

TABLE I.   BASIC STATISTICS OF THE FOUR DATASETS.

Dataset	Objects	Tags	Objects per tag	
			Maximum	Average
STACK OVERFLOW	47,668	437	6,113	234.93
FREECODE	39,231	243	9,615	545.08
ASK UBUNTU	37,354	346	6,169	234.03
ASK DIFFERENT	13,351	153	2,019	180.88

VII.   EXPERIMENTS AND RESULTS

In this section, we first present our experiment settings in Section VII-A. Our experiment results are then presented in Sections VII-B & VII-C. We discuss some interesting points in Section VII-D.

A. Experimental Setting

We evaluate ENTAGREC on four datasets: STACK OVERFLOW, ASK UBUNTU, ASK DIFFERENT (all three part of the Stack Exchange network), and FREECODE. STACK OVERFLOW is a Q&A site for software developers to post general programming questions. ASK DIFFERENT is a Q&A site related to Apple devices, e.g., iphone, ipad, mac. ASK UBUNTU is a Q&A site about Ubuntu. FREECODE is a site containing descriptions of many software projects.

Table I presents descriptive statistics of the four datasets. The STACK OVERFLOW and FREECODE datasets are obtained from Xia et al. and they have been used to evaluate TAGCOMBINE [40]. The ASK UBUNTU and ASK DIFFERENT datasets are new. We collect all questions in ASK UBUNTU and ASK DIFFERENT that are posted before April 2012. Following [40], to remove noise corresponding to tags that are assigned idiosyncratically, we filter out tags that are associated with less than 50 objects. These tags are less interesting since not many people use them, and thus they are less useful to be used as *representative tags* and recommending them does not help much in addressing the tag synonym problem addressed by tag recommendation studies. The numbers summarized in Table I are *after filtering*.

We perform a ten-fold cross validation [18] to evaluate our approach. We randomly split the dataset into ten subsamples. Nine of them are used as training data to train ENTAGREC and one subsample is used to test. We repeat the process ten times and use *Recall@k* as the evaluation metric. Unless otherwise stated, we set the values of $K_{Bayesian}$ and $K_{Frequentist}$ at 70.

Our evaluation involves two parts. First, we compare the effectiveness of ENTAGREC with those of TAGCOMBINE and the individual Bayesian and frequentist components that are integrated in ENTAGREC. TAGCOMBINE is the state-of-the-art tag recommendation approach, recently proposed by Xia et al [40]. Second, we perform sensitivity analyses to investigate the effect of different parameter values and different amounts of training data. We conduct all our experiments on a Windows 2008 server with 8 Intel®2.53GHz cores and 24GB RAM.

B. Experiment Results

We compare ENTAGREC with competing approaches: TAGCOMBINE proposed by Xia et al. [40] and the individual Bayesian and frequentist inference components (BIC and FIC) combined in ENTAGREC.

Table II presents the comparison between ENTAGREC, TAGCOMBINE and the individual Bayesian and frequentist

TABLE II.   *Recall@5* AND *Recall@10* FOR FOUR COMPETING
APPROACHES ENTAGREC, TAGCOMBINE [40], BAYESIAN AND
FREQUENTIST. THE HIGHEST VALUE IS TYPESET IN BOLDFACE.

Recall@5				
Dataset	ENTAGREC	TAGCOMBINE	Bayesian	Frequentist
STACK OVERFLOW	**0.805**	0.595	0.565	0.593
ASK UBUNTU	**0.815**	0.568	0.505	0.637
ASK DIFFERENT	**0.88**	0.675	0.523	0.713
FREECODE	**0.64**	0.639	0.391	0.545
Recall@10				
Dataset	ENTAGREC	TAGCOMBINE	Bayesian	Frequentist
STACK OVERFLOW	**0.868**	0.724	0.671	0.691
ASK UBUNTU	**0.876**	0.727	0.61	0.738
ASK DIFFERENT	**0.944**	0.821	0.633	0.827
FREECODE	0.753	**0.777**	0.506	0.626

TABLE III.   EFFECT SIZES (RECALL). E = ENTAGREC. T =
TAGCOMBINE, B = BAYESIAN, F = FREQUENTIST.

Recall@5			
Dataset	E vs. T	E vs. B	E vs. F
STACK OVERFLOW	0.60	0.70	0.53
ASK UBUNTU	0.76	0.91	0.46
ASK DIFFERENT	0.67	1.1	0.48
FREECODE	0.003	0.77	0.27
Recall@10			
Dataset	E vs. T	E vs. B	E vs. F
STACK OVERFLOW	0.46	0.63	0.49
ASK UBUNTU	0.49	0.85	0.4
ASK DIFFERENT	0.49	1.3	0.41
FREECODE	-0.08	0.80	0.40

inference components, presented in Section IV and V, respectively. We performed ten-fold cross-validation and evaluated the approaches in terms of *Recall@5* and *Recall@10*. ENTAG-REC achieves impressive improvements over TAGCOMBINE for the Stack Exchange datasets (more than 30% for *Recall@5* and more than 15% for *Recall@10*), and performs comparably to TAGCOMBINE on FREECODE. Averaging across the 4 datasets, ENTAGREC improves TAGCOMBINE in terms of *Recall@5* and *Recall@10* by 27.3% and 12.9% respectively. *Recall@5* and *Recall@10* of ENTAGREC are *always* higher than those of the individual components which demonstrates the value of combining them.

We note however, that *Recall@5* and *Recall@10* are defined as average (mean) values (cf. Definition 1). However, distribution of the

$$\frac{|Tags_i^{topK} \bigcap Tags_i^{correct}|}{|Tags_i^{correct}|}$$

values for software objects can hardly be expected to be symmetric, suggesting the use of mean to be problematic (cf. [37]). Indeed, for ENTAGREC and STACK OVERFLOW we have observed that the distribution is negatively skewed (skewness $\simeq -2$) and leptokurtic (kurtosis $\simeq 6.17$). Therefore, to obtain more insight into the recall values, we compare the distributions of *Recall@5* and *Recall@10* for ENTAGREC, TAGCOM-BINE, BIC and FIC by means of the recently-proposed multiple contrast test procedure $\widetilde{T}$ [22], [36], using 5% family-wise error rate. $\widetilde{T}$ is robust against unequal population variances and is applicable to different types of contrasts, including comparisons of multiple distributions (TAGCOMBINE, BIC and FIC) with a particular one (ENTAGREC), the so called Dunnett-type contrasts [12]. Applying $\widetilde{T}$ we observe that the recall values of ENTAGREC are higher than those of FIC and BIC, both for *Recall@5* and *Recall@10*, across the four datasets. The corresponding p-values were too small to be computed exactly (< 0.01), showing the significance of the differences in the recall values achieved by the different approaches. For the Stack Exchange datasets (STACK OVERFLOW, ASK UBUNTU, ASK DIFFERENT) $\widetilde{T}$ establishes that ENTAGREC outperforms TAGCOMBINE both for *Recall@5* and *Recall@10*. For FREE-CODE, the difference between ENTAGREC and TAGCOMBINE in terms of *Recall@5* is not statistically significant. However, TAGCOMBINE significantly outperforms ENTAGREC in terms of *Recall@10* but the absolute difference is small (0.024).

To investigate if the differences in the recall values are *substantial*, we also compute Cohen's d [10] which measures effect size. The results are shown in Table III. Cohen defined an effect size of 0.2, 0.5, 0.8 to be small, medium, and large
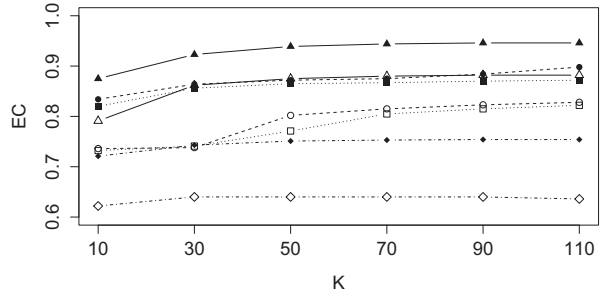

Fig. 5.   Results for different $K_{Bayesian}$ and $K_{Frequentist}$ for $K_{Bayesian} = K_{Frequentist}$: *Recall@5* (empty symbols), *Recall@10* (filled symbols), STACK OVERFLOW (squares), ASK UBUNTU (circles), ASK DIFFERENT (triangles) and FREECODE (diamonds).

respectively [10]. If the effect size is close to 0, it means that the difference is not substantial. From the results we can conclude that ENTAGREC substantially outperforms TAGCOM-BINE, BIC, and FIC on STACK OVERFLOW, ASK UBUNTU, and ASK DIFFERENT datasets (with close-to-medium to large effect sizes). For the FREECODE dataset, ENTAGREC also substantially outperforms BIC and FIC (with small to large effect sizes). However, the difference between the performance of ENTAGREC and TAGCOMBINE on FREECODE is not substantial (effect sizes close to zero).

C. Sensitivity Analyses

We perform two different sensitivity analyses. We start by varying parameters $K_{Bayesian}$ and $K_{Frequentist}$ of ENTAGREC, and proceed with varying the size of the training data.

To understand the impact of varying the ENTAGREC-parameters $K_{Bayesian}$ and $K_{Frequentist}$ on the recall values, we vary both parameters over $\{10, 30, 50, 70, 90, 110\}$ and compute *Recall@5* and *Recall@10*. For the sake of simplicity we set $K_{Bayesian} = K_{Frequentist}$. Our main goal is to investigate whether the performance of ENTAGREC is robust across a wide range of parameter values for $K_{Bayesian}$ and $K_{Frequentist}$.

Figure 5 shows that the recall values of ENTAGREC exhibit an increasing trend followed by stabilization. While the best recall is achieved for the largest values of the parameters, the time needed to train and recommend tags increases as well. We note that the results of ENTAGREC remain relatively stable across a wide range of parameters $K_{Bayesian}$ and $K_{Frequentist}$. In this work, we choose the value where stabilization can be observed both for *Recall@5* and *Recall@10* across four datasets i.e., 70, as the default parameters of ENTAGREC.

As opposed to explicit parameters $K_{Bayesian}$ and $K_{Frequentist}$ that influence the results of ENTAGREC, the amounts of
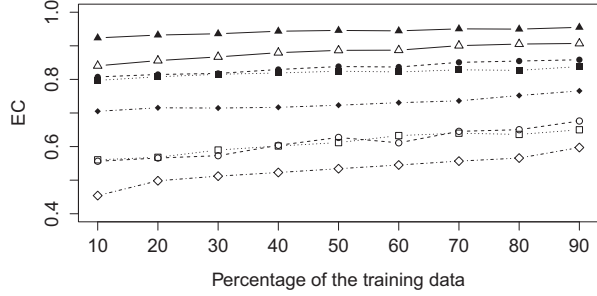
Fig. 6. Recall values (y) decreases gracefully as the percentage of the training data (x) decreases: : *Recall@5* (empty figures), *Recall@10* (black figures), STACK OVERFLOW (squares), ASK UBUNTU (circles), ASK DIFFERENT (triangles) and FREECODE (diamonds).

training and test data implicitly affect the results. We would like to understand the impact of the training and test data on the results of the algorithm. Our main goal is to investigate whether ENTAGREC can work reasonably well when only a limited amount of training data is available.

Hence, we vary the amount of training and test data, from 10% to 90% training data, on each one of the four datasets. The training data is randomly selected and the remaining part of the data is used as test data. Figure 6 shows that the values of *Recall@5* (empty symbols) and *Recall@10* (filled symbols) decrease gracefully as the percentage of training data decreases. When we reduce the training data from 90% to 50% the average reduction in *Recall@10* and *Recall@5* are 5.5% and 15.7%, respectively. Also, when we reduce the training data from 90% to 10% the average reduction in *Recall@10* and *Recall@5* are 5.9% and 19.2%, respectively.

D. Discussion

Example. Figure 7 shows a software object from STACK OVERFLOW with the `ruby` and `rdoc` tags. TAGCOMBINE cannot infer any of the tags. On the other hand ENTAGREC can infer all tags. This is one of the many examples where the performance of ENTAGREC is better than TAGCOMBINE.
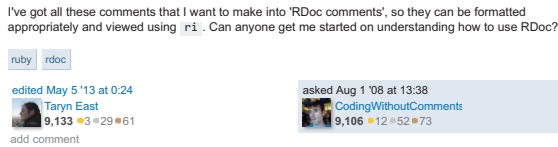

Fig. 7. ENTAGREC correctly suggests tags `ruby` and `rdoc` for this STACK OVERFLOW question, while TAGCOMBINE does not.

Precision@k Results. Aside from *Recall@k*, *Precision@k* has also been used to evaluate information retrieval techniques. Definition 2 defines *Precision@k*. In this paper, we focus on *Recall@k* as the evaluation metric. A similar decision was made by past tag recommendation studies [40], [41]. This is the case as the number of tags that are attached to an object is often small (much less than K). Thus, the value of *Precision@k* is often very low and is not meaningful.

Definition 2: Consider a set of n software objects. For each object o_i, let the set of its correct (i.e., ground truth) tags be

TABLE IV. *Precision@5* AND *Precision@10* FOR FOUR COMPETING APPROACHES ENTAGREC, TAGCOMBINE [40], BAYESIAN AND FREQUENTIST. THE HIGHEST VALUE IS TYPESET IN BOLDFACE.

Precision@5				
Dataset	ENTAGREC	TAGCOMBINE	Bayesian	Frequentist
STACK OVERFLOW	**0.346**	0.221	0.232	0.258
ASK UBUNTU	**0.358**	0.251	0.212	0.282
ASK DIFFERENT	**0.369**	0.278	0.212	0.298
FREECODE	**0.382**	0.381	0.230	0.322
Precision@10				
Dataset	ENTAGREC	TAGCOMBINE	Bayesian	Frequentist
STACK OVERFLOW	**0.187**	0.151	0.141	0.151
ASK UBUNTU	**0.193**	0.158	0.131	0.165
ASK DIFFERENT	**0.200**	0.173	0.130	0.175
FREECODE	0.240	**0.249**	0.153	0.200

TABLE V. EFFECT SIZES (PRECISION). E = ENTAGREC. T = TAGCOMBINE, B = BAYESIAN, F = FREQUENTIST.

Precision@5			
Dataset	E vs. T	E vs. B	E vs. F
STACK OVERFLOW	0.78	0.62	0.40
ASK UBUNTU	0.55	0.78	0.33
ASK DIFFERENT	0.49	0.87	0.33
FREECODE	0.005	1.0	0.32
Precision@10			
Dataset	E vs. T	E vs. B	E vs. F
STACK OVERFLOW	0.38	0.50	0.33
ASK UBUNTU	0.35	0.64	0.24
ASK DIFFERENT	0.28	0.75	0.24
FREECODE	-0.06	0.65	0.26

$Tags_i^{correct}$. Also, let $Tags_i^{topK}$ be the top-k ranked tags that are recommended by a tag recommendation approach for o_i. Average *Precision@k* for n is given by:

$$Precision@k = \frac{1}{n} \sum_{i=1}^{n} \frac{|Tags_i^{topK} \bigcap Tags_i^{correct}|}{|Tags_i^{topK}|}$$

Still, for completeness sake, we show the *Precision@k* results in Table IV. The results show that ENTAGREC outperforms TAGCOMBINE, BIC, and FIC on all datasets in terms of *Precision@5*. In terms of *Precision@10*, ENTAGREC outperforms BIC and FIC on all datasets; ENTAGREC also outperforms TAGCOMBINE on three out of the four datasets. The difference between *Precision@10* of ENTAGREC and TAGCOMBINE is small (i.e., 0.009). We also performed the T test and found that in terms of *Precision@5*, ENTAGREC significantly outperforms TAGCOMBINE, BIC and FIC. Also, in terms of *Precision@10*, ENTAGREC significantly outperforms BIC and FIC on all datasets and TAGCOMBINE on STACK OVERFLOW, ASK UBUNTU, and ASK DIFFERENT. For FREECODE, TAGCOMBINE significantly outperforms ENTAGREC in terms of *Precision@10*.

To investigate if the differences in the precision values are substantial, we also compute Cohen's d which measures effect size. The results are shown in Table V. From the results we can conclude that ENTAGREC substantially outperforms TAGCOMBINE, BIC, and FIC on the STACK OVERFLOW, ASK UBUNTU, and ASK DIFFERENT datasets (with small to large effect sizes). For the FREECODE dataset, ENTAGREC also substantially outperforms BIC and FIC (with small to large effect sizes). However, the difference between the performance of ENTAGREC and TAGCOMBINE on FREECODE is not substantial (effect sizes close to zero).

Efficiency. We find that ENTAGREC runtimes for the training and deployment phases are reasonable. ENTAGREC's training time can mostly be attributed to training an L-LDA model in

the Bayesian inference component of ENTAGREC, which never exceeds 18 minutes (measured on the STACK OVERFLOW dataset, the largest of the four). The Frequentist inference component is much faster; its runtime never exceeds 40 seconds (measured on the STACK OVERFLOW dataset). In the deployment phase, the average time ENTAGREC takes to recommend a tag never exceeds 0.128 seconds.

Threats to Validity. Threats to external validity relate to the generalizability of our results. We have analyzed four popular software information sites and more than 130,000 software objects. In the future, we plan to reduce this threat further by analyzing even more software objects from more software information sites. As a threat to internal validity, we assume that the data in the software information sites are correct. To reduce the threat we only used older data—assuming people correct wrongly/poorly assigned tags. Also, two of our datasets (i.e., STACK OVERFLOW and FREECODE) were used in a past study [40]. We use a lot of data and only consider tags that are used to label at least 50 objects to further reduce the impact of noise. Furthermore, manual inspection of a random sample of 100 Stack Overflow objects (questions) revealed that only 1 had a clearly irrelevant tag (out of a total of 3 tags for that object), and for 19 others we couldn't accurately determine the relevancy of some of their tags (typically one out of 3 or more tags per object).

Threats to construct validity relate to the suitability of our evaluation metrics. We have used *Recall@k* and *Precision@k* to evaluate our proposed approach ENTAGREC in comparison with other approaches. These measures are standard information retrieval measures used by prior tag recommendation studies, e.g., [1], [40], [41]. We have also performed statistical test and effect size test to check if the differences in *Recall@k* and *Precision@k* are significant and substantial. Thus, we believe there is little threat to construct validity.

VIII. RELATED WORK

Tag Recommendation: Al-Kofahi et al. proposed TAGREC which recommends tags in work item systems (e.g., IBM Jazz) [1]. There are a number of studies from the data mining research community, that recommend tags for social media sites like Twitter, Delicious, and Flickr [19], [30], [41]. Among these studies, the work by Zangerle et al. is the latest approach to recommend hashtags for short messages in Twitter [41]. Xia et al. proposed TAGCOMBINE, which combines three components: a multi-label ranking component, a similarity based ranking component, and a tag-term based ranking component [40]. The multi-label ranking component employs a multi-label classification algorithm (i.e., binary relevance method with naive Bayes as the underlying classifier) to predict the likelihood of a tag to be assigned to a software object. The similarity based-ranking component predicts the likelihood of a tag (to be assigned to a software object) by analyzing the tags that are given to the top-k most similar software objects that were tagged before. The tag-term based ranking component predicts the likelihood of a tag (to be assigned to a software object) by analyzing the number of times a tag has been used to tag a software object containing a term (i.e., a word) before. Xia et al. have shown that TAGCOMBINE outperforms TAGREC and Zangerle et al.'s approach in recommending tags in software information sites.

The closest work to ours is TAGCOMBINE proposed by Xia et al. which is also the state-of-the-art work [40]. There are a number of technical differences between our approach, named ENTAGREC, and TAGCOMBINE. ENTAGREC combines two components: a Bayesian inference component that employs Labeled LDA (BIC), and an enhanced frequentist inference component that removes unrelated words with the help of a parts-of-speech (POS) tagger, and finds associated tags with a spreading activation algorithm (FIC). Our BIC is related to the multi-label ranking component of TAGCOMBINE since both of them employ Bayesian inference. The multi-label ranking component of TAGCOMBINE constructs many one-versus-rest Naive Bayes classifiers, one for each tag. Each Naive Bayes classifier simply predicts the likelihood of a software object to be assigned a particular tag. In ENTAGREC, we construct only one classifier which is a *mixture model* that considers all tags together. Mixture models have been shown to outperform one-versus-rest traditional multi-label classification approaches [15], [26], [27]. Also, our FIC removes unrelated words (using POS tagger) and finds associated tags (using spreading activation) while none of the three components of TAGCOMBINE perform these. We have compared our approach with TAGCOMBINE, on four datasets: STACK OVERFLOW, ASK UBUNTU, ASK DIFFERENT, and FREECODE. We show that our approach outperforms TAGCOMBINE on three datasets (i.e., STACK OVERFLOW, ASK UBUNTU, ASK DIFFERENT), and performs as well as TAGCOMBINE on one dataset (i.e., FREECODE).

Tagging in Software Engineering: The need for automatic tag recommendation has been recognized both by practitioners [20], [21], [39] and by researchers. Aside from tag recommendation studies mentioned above, there are several software engineering studies that also analyze tagging and leverage tags for various purposes. Treude et al. performed an empirical study on the impact of tagging on a large project with 175 developers over a two years period [34]. Wang et al. analyzed tags of projects in FREECODE, inferred the semantic relationships among the tags, and expressed the relationships as a taxonomy [38]. Thung et al. detected similar software applications using software tags [32]. Gottipati et al. automatically tagged posts in software forums with 7 predefined tags to help improve a search engine that finds relevant answers from forum threads [17].

IX. CONCLUSION AND FUTURE WORK

In this work, we propose a novel approach to recommend tags to software information sites. Our approach, named EN-TAGREC, learns from tags of historical software objects to infer tags of new software objects. To recommend tags, EN-TAGREC performs a combination of Bayesian and frequentist inferences, which are two opposite yet complementary lines of thought in the statistics community. To perform Bayesian inference, we map the tag recommendation problem to a supervised topic mining problem, and make use of Labeled Latent Dirichlet Allocation (L-LDA) which performs statistical analysis leveraging the Bayes rule. To perform frequentist inference, we count the number of nouns and noun phrases that are used in training software objects which are assigned a particular tag, and leverage the relationships among tags. We make use of a part-of-speech tagger and the spreading activation algorithm to perform frequentist inference. ENTAGREC composes the two inferences by assigning weights to each of them; the weights are inferred by finding the best weights that optimizes the performance of ENTAGREC on a

training dataset. We evaluated the performance of ENTAGREC on four datasets, STACK OVERFLOW, ASK UBUNTU, ASK DIFFERENT, and FREECODE, which contain 47,688, 39,231, 37,354, and 13,351 software objects, respectively. We found that that ENTAGREC achieves *Recall@5* scores of 0.805, 0.815, 0.88 and 0.64, and *Recall@10* scores of 0.868, 0.876, 0.944 and 0.753, on STACK OVERFLOW, ASK UBUNTU, ASK DIFFERENT, and FREECODE, respectively. In terms of *Recall@5* and *Recall@10*, averaging across the 4 datasets, ENTAGREC improves TAGCOMBINE [40], which is the state of the art approach, by 27.3% and 12.9% respectively.

As future work, we plan to reduce the threats to validity by experimenting with more software objects from more software information sites. Also, we plan to perform a qualitative study to better understand the nature of the inferred tags and a sensitive study to investigate how robust is the proposed approach against poor selection of parameters. Furthermore, we plan to improve the *Recall@5* and *Recall@10* of ENTAGREC further by investigating cases where ENTAGREC is inaccurate, and by building a more sophisticated machine learning solution.

REFERENCES

[1] J. M. Al-Kofahi, A. Tamrawi, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Fuzzy set approach for automatic tagging in evolving software. In *ICSM*, pages 1–10, 2010.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, Oct. 2002.

[3] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In *ICSE*, pages 95–104, 2010.

[4] P. Baldi, C. V. Lopes, E. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *OOPSLA*, pages 543–562, 2008.

[5] M. J. Bayarri and J. O. Berger. The interplay of Bayesian and frequentist analysis. *Statistical Science*, 19(1):58–80, 2004.

[6] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *JMLR*, 13:281–305, 2012.

[7] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *JMLR*, pages 993–1022, 2003.

[8] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *CHI*, pages 1589–1598. ACM, 2009.

[9] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella. Improving IR-based traceability recovery via noun-based indexing of software artifacts. *Journal of Software: Evolution and Process*, 25(7):743–762, 2013.

[10] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates., 1988.

[11] F. Crestani. Application of spreading activation techniques in information retrieval. *Artif. Intell. Rev.*, 11(6):453–482, 1997.

[12] C. W. Dunnett. A multiple comparison procedure for comparing several treatments with a control. *Journal of American Statistical Association*, 50(272):1096–1121, 1955.

[13] B. Efron. Why isn't everyone a Bayesian? *The American Statistician*, 40(1):1–5, Feb. 1986.

[14] A. Gelman, J. Carlin, H. Stern, and D. Rubin. *Bayesian Data Analysis*. CRC Press, 2003.

[15] N. Ghamrawi and A. McCallum. Collective multi-label classification. In *CIKM*, pages 195–200, 2005.

[16] S. A. Golder and B. A. Huberman. Usage patterns of collaborative tagging systems. *Journal of Information Science*, 32(2):198–206, Apr. 2006.

[17] S. Gottipati, D. Lo, and J. Jiang. Finding relevant answers in software forums. In *ASE*, pages 323–332, 2011.

[18] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 2011.

[19] R. Jäschke, L. B. Marinho, A. Hotho, L. Schmidt-Thieme, and G. Stumme. Tag recommendations in folksonomies. In *PKDD*, 2007.

[20] jmac. Select and display 'suggested tags' for all posts based on related questions (or other logic), Sept. 2013. http://meta.stackexchange.com/q/196702/182512.

[21] Jud.Her. Tag recommendations for Stack Overflow, Apr. 2011. http://meta.stackexchange.com/q/88611/182512.

[22] F. Konietschke, L. A. Hothorn, and E. Brunner. Rank-based multiple test procedures and simultaneous confidence intervals. *Electronic Journal of Statistics*, 6:738–759, 2012.

[23] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information & Software Technology*, 52(9):972–990, 2010.

[24] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. D. Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *ICSE*, pages 522–531, 2013.

[25] M. F. Porter. An algorithm for suffix stripping. In *Readings in information retrieval*, pages 313–316. Morgan Kaufmann, 1997.

[26] A. Puurula. Mixture models for multi-label text classification. In *10th New Zealand Computer Science Research Student Conference*, 2011.

[27] D. Ramage, D. Hall, R. Nallapati, and C. D. Manning. Labeled lda: a supervised topic model for credit attribution in multi-labeled corpora. In *EMNLP '09*, pages 248–256, 2009.

[28] F. I. Samaniego. *A Comparison of the Bayesian and Frequentist Approaches to Estimation*. Series in Statistics. Springer, 2010.

[29] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *MSR*, 2013.

[30] B. Sigurbjörnsson and R. van Zwol. Flickr tag recommendation based on collective knowledge. In *WWW '08*, pages 327–336, 2008.

[31] M.-A. Storey, C. Treude, A. van Deursen, and L.-T. Cheng. The impact of social media on software engineering practices and tools. In *FoSER '10*, pages 359–364, 2010.

[32] F. Thung, D. Lo, and L. Jiang. Detecting similar applications with collaborative tagging. In *ICSM*, pages 600–603, 2012.

[33] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *HLT-NAACL*, 2003.

[34] C. Treude and M.-A. Storey. How tagging helps bridge the gap between social and technical aspects in software development. In *ICSE '09*, pages 12–22, 2009.

[35] B. Vasilescu, A. Serebrenik, P. T. Devanbu, and V. Filkov. How social Q&A sites are changing knowledge sharing in open source software communities. In *CSCW*, pages 342–354, 2014.

[36] B. Vasilescu, A. Serebrenik, M. Goeminne, and T. Mens. On the variation and specialisation of workload - a case study of the Gnome ecosystem community. *Empirical Software Engineering*, 19(4):955–1008, 2014.

[37] B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand. By no means: A study on aggregating software metrics. In *2nd International Workshop on Emerging Trends in Software Metrics*, WETSoM, pages 23–26. ACM, 2011.

[38] S. Wang, D. Lo, and L. Jiang. Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging. In *ICSM*, pages 604–607, 2012.

[39] D. Warbox. Auto-tagging, July 2009. http://meta.stackoverflow.com/questions/1377/auto-tagging.

[40] X. Xia, D. Lo, X. Wang, and B. Zhou. Tag recommendation in software information sites. In *MSR '13*, pages 287–296, 2013.

[41] E. Zangerle, W. Gassler, and G. Specht. Using tag recommendations to homogenize folksonomies in microblogging environments. In *SocInfo'11*, pages 113–126, 2011.